Multitasking

2501ICT/7421ICTNathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

1. Introduction to Multitasking
   - Overview
   - Multitasking Introduction
2. Tasks
   - Task Models
   - Processes and Threads
3. Concurrency and Synchronisation
   - Concurrency
   - Task Synchronisation

Overview

- Multitasking Basics
- Process/Thread Life Cycle
- Creating and Starting Threads
- Creating and Starting Processes
- Concurrency
- Semaphores

Processes

- Running Program: a *Process*
  - when started, there is just one Process
  - Sequential Program
    - start
    - sequence of statements (Instructions)
    - program counter (Instruction Counter)
    - end
- Spawning a new Process
  - Do more than one thing at a time

Multitasking

- Run more than one Process
  - multiple processors (Multiprocessing)
  - single processor (Timesharing)
- Non-Preemptive Multitasking
  - Process does not get interrupted
  - Yields Processor when waiting (e.g. File I/O)
- Preemptive Multitasking
  - Process loses CPU after using up its Slice of Time (Time-Slicing)

Non-Preemptive Multitasking

+ Easier to implement in an OS
– Processes must yield CPU
    - Cooperative Multitasking
    - what happens if a Process doesn't cooperate?
        - it uses all CPU, no other Process can run
        - whole system can hang
- OS Examples
    - Windows <= 3.x, MacOS <= 9.x
    - Some embedded systems

Preemptive Multitasking

– More Complex
  - requires Time Slice Controller
  - access to shared resources: Synchronisation

+ Allows “simultaneous” Processes
  - appear to be all running at the same time

- OS Examples
  - Windows >= 95, MacOS X, Linux, BSD, Unix
  - Modern embedded Systems

Why Multitasking?

- Run multiple programs at a time
  - appear to be running simultaneously
- Run multiple Threads/Processes within the same program
  - do a number of things concurrently

→ Browser: scroll pages during download

→ Multimedia: play sound and video at the same time

Processes

- Modern Operating Systems offer Memory Protection
- Separate (writable) Data Space for each Process

+ One Program cannot overwrite other Processes' Memory
+ If one Program crashes, other Programs and the Operating System can continue

– Process (Context) Switching Overhead (MMU)
– Difficult to Share Data among Processes

Solutions

- Data Sharing
  - Shared Memory
  - Message Queues
  - Pipes
  - Sockets (“networked Pipes”)
  - Using Threads instead of Processes
- Context Switch Overhead
  - Threads (lightweight Processes)

Threads

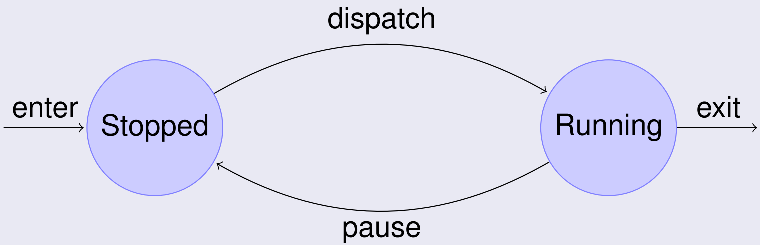
- Common Memory
  - → among all Threads belonging to a Process
  - context switching is quick
- A Thread can overwrite other Threads' memory
  - easy data sharing
  - unwanted side-effects (inconsistency, memory corruption)

Tasks

- Threads
  - can usually be *switched quicker* than Processes
  - do *not have memory protection*
  - intrinsically shared data
- Processes
  - have more *context-switching overhead*
  - are usually *protected* from one another
  - require *explicitly shared memory*

Task States

Two State Task Model

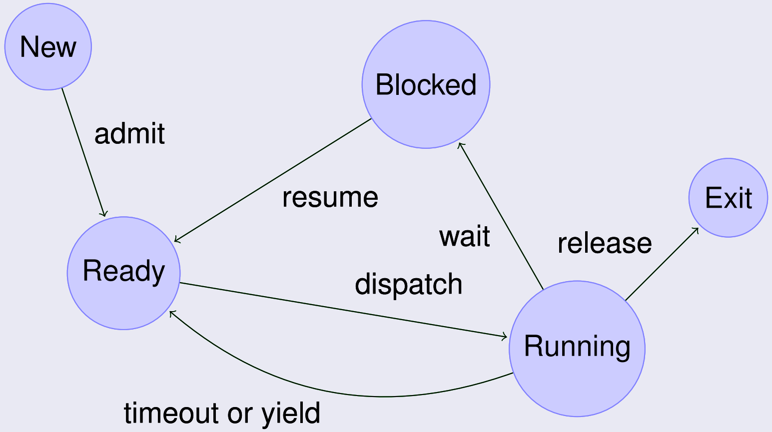


Two State Model Problems

- A Task is not always ready
  - it could be blocked while waiting for ...
    - → ... user input, a hardware device, data from another Process, etc.
- When the Task is ready ...
  - → the CPU could be fully utilised by another Task
- Management Overhead
  - *New* and *Exit* states

Five State Model

Five State Task Model



Scheduling

- The Scheduler enqueues Tasks
  - Ready Queue contains all scheduled Tasks
  - different algorithms determine priority
    - → FCFS, Round Robin, Fair Share, Shortest Process Next (SPN), Shortest Remaining Time (SRT), . . .
- The Dispatcher
  - runs the first Task on the Ready Queue
  - as long as tasks and CPUs are available
  - Timeout or Yield returns CPU to the Dispatcher

Processes in C

- `fork()`
  - creates a new Child Process
  - Parent and Child execute exactly the same Code

    → the return value of `fork()` is used to distinguish between Parent (old Process) and Child (new Process)
- `wait()`
  - waits for Child to exit
  - collects the Child's status
  - needs to be called by Parent at the end

Forking a new Process

Example (prints `parent child` or `child parent`)

```
#include <unistd.h>                          // required for fork()
#include <sys/wait.h>                        // required for wait()
#include <stdio.h>                           // required for perror()
#include <stdlib.h>                          // required for exit()

int main(int argc, char *argv[])            // standard main() function
{
        int status = EXIT_SUCCESS;          // child process exit status
        pid_t pid = fork();                 // fork child process

        switch (pid)                        // check fork() return value
        {
                case -1:                    // an error occurred
                    perror("fork");         // print an error message
                    status=EXIT_FAILURE;    // exit with failure status
                break;
                case 0:                     // this is the child process
                    printf("child\n");      // execute child code
                break;
                default:                    // parent process, pid = child ID
                    printf("parent\n");     // execute parent code
                    wait(&status);          // wait for child to exit
        }
        return status;                      // return the child status
}
```

Child Processes

- Get a unique Process ID (`pid`)
- Inherit from their Parent Process . . .
  - → all variables and open files
- Run in a separate, protected memory area
- Often used to run external program
  - → `exec()` system calls in C, Objective-C, or C++
  - → `NSTask` class in Objective-C

NSTask example

Example (list the current directory using ls)

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
  NSAutoreleasePool *pool = [NSAutoreleasePool new];
  NSString *cmd = @"/bin/ls";                                    // "ls" command
  NSArray *args = [NSArray arrayWithObjects: @"-als", nil];       // -als args

  NSTask *task = [NSTask launchedTaskWithLaunchPath: cmd          // run command
                                          arguments: args];       // with arguments
  /*
   * "ls -als" now runs in the background, so we can do something else
   * in the meantime, then wait until the external task has exited
   */
  [task waitUntilExit];                                          // wait for ls

  int status = [task terminationStatus];                         // get exit value
  printf("Task returned %d\n", status);                          // and print

  [pool release];

  return EXIT_SUCCESS;
}
```

execl() example

Example (list the current directory using ls)

```
#include <unistd.h>                            // required for fork()/execl()
#include <sys/wait.h>                          // required for wait()
#include <stdio.h>                             // required for perror()
#include <stdlib.h>                            // required for exit()

int main(int argc, char *argv[])              // standard main() function
{
  pid_t pid;                                   // the child process ID
  int status = EXIT_SUCCESS;                   // child process exit status

  pid = fork();                                // fork child process
  if (pid == -1) {                             // an error  occurred
      perror("fork");                          // print an error message
      status=EXIT_FAILURE;                     // exit with failure status
  }
  else if (pid == 0) {                         // this is the child process
      status = execl("/bin/ls", "ls", "-als", NULL);    // execute "ls -als"
  } else {                                     // parent process, pid = child ID
      wait(&status);                           // wait for child to exit
      printf("child returned %d\n", status);   // print child status
  }

  return status;                               // return the child status
}
```

Threads in C

- `pthread_create()`
  - spawns a new thread
  - takes a function as an argument

    $\rightarrow$ new thread will call this function
- `pthread_exit()`
  - exits current thread (like `exit()` for processes)
- `pthread_join()`
  - waits for thread to exit (like `wait()` for processes)

Spawning a new Thread in C

Example (spawning a thread)

```
#include <pthread.h>                           // required for threads
#include <stdio.h>                             // required for printf()
#include <stdlib.h>                            // required for exit()

void *child(void *arg)                        // child function
{
  printf("%s\n", arg);                        // print argument
  return "okay";                              // exit status
}

int main(int argc, char *argv[])              // standard main() function
{
  pthread_t tid;                              // child thread ID
  char *arg = "child";                        // argument passed to child
  void *status;                               // child status

  if (pthread_create(&tid, NULL, child, arg) != 0) {     // spawn child, check error
    perror("error creating child");           // print error message
    return EXIT_FAILURE;                      // didn't work --> exit
  }
  printf("parent\n");                         // execute some parent process code
  pthread_join(tid, &status);                 // wait for child
  printf("child said: %s\n", status);          // print child status

  return EXIT_SUCCESS;
}
```

Threads in Objective-C

- `NSThread` class
  - allows a method (selector) on an object to be invoked on a child thread
  - `[NSSThread detachNewThreadSelector: sel toTarget: t withObject: obj]`
    - launches (detaches) new thread
- `+exit`
  - class method that exits the current thread
- `+currentThread`
  - returns the current thread object

Threads in C++-11

- `std::thread` class
  - allows a C++ function to be called on a child thread
  - `std::thread my_thread(some_function);`
    - constructor launches new thread
- `join()`
  - instance method that waits for the thread

Concurrency Problems

- Two Tasks accessing common resources (e.g. memory)
  - → no problem as long as both tasks only read
  - what happens if one task writes while the other task reads?
  - what happens if both tasks try writing?

→ Let's look at some examples!

Concurrency Example (1)

Example (two tasks modifying shared data)

```c
int shared = 0;

void task1(void)
{
        shared = 1;
}
```

```c
extern int shared;

void task2(void)
{
        shared = 2;
}
```

- No concurrency problem!
  - `shared` is either 0, 1, or 2

→ Both tasks use Atomic Operations

Concurrency Example (2)

Example (two tasks modifying shared data)

```
int shared = 0;

void task1(void)
{
        shared++;
        shared++;
}
```

```
extern int shared;

void task2(void)
{
        shared += 2;

}
```

- Inconsistencies can occur!
  - tasks can interrupt each other at critical points
  - *Read-Modify-Write* operations are not Atomic
  - ⇒ `shared` can suddenly end up with an odd value

Avoiding Inconsistencies

- Always use Atomic Actions
  - not always possible for certain operations
  - hard to tell if an operation is atomic

    → depends on compiler and system implementation
- Protect Critical Regions
  - use synchronisation constructs before accessing shared resources

  → transforms operations into atomic actions

Mutual Exclusion, Attempt #1

Example (turn-based mutual exclusion)

```c
int turn = 0;
int shared = 0;

void task1(void)
{
        while (turn != 0)
                ; // do nothing

        // critical section
        shared++;
        shared++;

        turn = 1;
}
```

```c
extern int turn;
extern int shared;

void task2(void)
{
        while (turn != 1)
                ; // do nothing

        // critical section
        shared += 2;
        // end critical section

        turn = 0;
}
```

Analysis of Attempt #1

- Guarantees Mutual Exclusion
- Drawbacks
  - tasks are forced to strictly alternate their use of the shared resource
    - $\Rightarrow$ pace is dictated by the slower process
  - if one Task fails even outside the critical region, the other Task is stuck forever
  - Waiting Task consumes 100% CPU time
    - $\rightarrow$ Busy Waiting

Attempt #2

Example (flag-based mutual exclusion)

```
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
        while (flag[1])
                ; // do nothing

        flag[0] = TRUE;
        // critical section
        shared++;
        shared++;
        flag[0] = FALSE;
}
```

```
extern int flag[2];
extern int shared;

void task2(void)
{
        while (flag[0])
                ; // do nothing

        flag[1] = TRUE;
        // critical section
        shared += 2;
        // end critical section
        flag[1] = FALSE;
}
```

Analysis of Attempt #2

- Task failing outside Critical Section
  - → no longer affects the other task!
- Mutual Exclusion not guaranteed:
  - Task 0 enters and exits `while()` because `flag[1]` is FALSE
  - Task 1 enters and exits `while()` because `flag[0]` is FALSE
  - both set their flags and enter critical section!
    - ⇒ flags are set too late!

Attempt #3

Example (setting flags first)

```
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
        flag[0] = TRUE;
        while (flag[1])
                ; // do nothing

        // critical section
        shared++;
        shared++;
        flag[0] = FALSE;
}
```

```
extern int flag[2];
extern int shared;

void task2(void)
{
        flag[1] = TRUE;
        while (flag[0])
                ; // do nothing

        // critical section
        shared += 2;
        // end critical section
        flag[1] = FALSE;
}
```

Analysis of Attempt #3

- Mutual Exclusion guaranteed
  - only one Task enters critical section at a time
- Deadlock can occur:
  - both tasks set their flags to `TRUE`
  - both tasks enter their `while()` loops and `wait indefinitely` for the other task to clear its flag!
  - no task will ever be able to do anything useful again.

Attempt #4

Example (backing off)

```
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
        flag[0] = TRUE;
        while (flag[1]) {
                flag[0] = FALSE;
                // delay a bit
                flag[0] = TRUE;
        }
        // critical section
        shared++;
        shared++;
        flag[0] = FALSE;
}
```

```
extern int flag[2];
extern int shared;

void task2(void)
{
        flag[1] = TRUE;
        while (flag[0]) {
                flag[1] = FALSE;
                // delay a bit
                flag[1] = TRUE;
        }
        // critical section
        shared += 2;
        // end critical section
        flag[1] = FALSE;
}
```

Analysis of Attempt #4

- Close to a correct solution
  - mutual exclusion guaranteed, no Deadlock
- Livelock can occur:
  - both tasks set their flags to `TRUE`
  - both tasks check the their task's flag (`TRUE`)
  - both tasks release their flag and start again

  → endless loop grabbing and releasing their flag, *consuming 100% of (useless) CPU time*

Peterson's Algorithm

Example (backing off)

```
int flag[2] = {FALSE, FALSE};
int turn = 0;

void task1(void)
{
        flag[0] = TRUE, turn = 1;
        while (flag[1] && turn==0)
                ; // do nothing

        // critical section
        shared++;
        shared++;

        flag[0] = FALSE;
}
```

```
extern int flag[2];
extern int turn;

void task2(void)
{
        flag[1] = TRUE, turn = 0;
        while (flag[0] && turn==1)
                ; // do nothing

        // critical section
        shared += 2;
        // end critical section

        flag[1] = FALSE;
}
```

Peterson's Algorithm (2)

- Correct solution
  - mutual Exclusion, no Dead-/Livelocks
- Not a generic solution
  - works only for two tasks
  - → still uses Busy Waiting
- Solution: Hardware and/or OS-Support
  - atomic Test-And-Set (TAS) CPU instructions
  - blocking a task w/o consuming CPU time

Semaphores

- Simple Signalling Mechanism
  - synchronisation of multiple Tasks
- Shared Integer Variable
  - usually initialised to nonnegative value
  - `Wait()` operation: `P()`
    - block task while semaphore ≤ 0, decrement value
  - `Signal()` operation: `V()`
    - increment value, unblock task(s) on waiting queue

Semaphore Algorithm

Semaphore Operations

```
int semaphore = 1;

P()
{
        while (semaphore <= 0)
                BLOCK;

        semaphore--;
}
```

```
extern int semaphore;

V()
{
        semaphore++;

        WAKEUP;
}
```

- `P()` and `V()` cannot be interrupted!
- `BLOCK` enqueues a Task on the waiting queue
- `WAKEUP` removes the first Task from the waiting queue

Semaphore Advantages

→ Flexibility!

- Multiple tasks
  - more than two tasks can be synchronised
- If initialised to an `n > 1`
  - *n* tasks can enter critical region!
- If initialised to an `n < 1`
  - $-n+1$ `V()` operations are required before first task can enter critical region!

Semaphores in C

- Create and initialise a Semaphore
  - `sem_open()`
  - → `sem_t *s = sem_open("mysemaphore", O_CREAT, 0600, 1);`
- `P()`
  - `sem_wait()`
- `V()`
  - `sem_post()`

Task Synchronisation

- Semaphores
  - means for protecting critical regions
  - flexible method, handling more than one task
- `NSLock` Objective-C class
  - simple binary semaphore (0 and 1 values only)
  - → always initialised to 1
  - `-lock`
    - `P()` operation (set semaphore to 0)
  - `-unlock`
    - `V()` operation (set semaphore to 1)
    - → needs to be called by the task that called `lock`
    - → `lock` must have been called before unlock