Midterm #1

CMSC 412
Operating Systems
Fall 2005

October 17, 2005

Guidelines

This exam has 10 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

You may avail yourself of the punt rule. If you write down punt for any part of a question with a specifically-assigned point value, you will earn 1/5 of the points for that question (rounded down).

Use good test-taking strategy: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

Question	Points	Score
1	25	
2	15	
3	20	
4	25	
5	15	
Total	100	

1. (Scheduling, 25 points)

   (a) (4 points) What is meant when saying that a scheduling algorithm could *result in starvation?*

   Answer:

   That it is possible for a process to be runnable but never scheduled.

   (b) (6 points) Clearly mark the algorithms listed below that could result in starvation. (You may explain why if you wish.) You can assume that each process will use the CPU for a finite burst before performing I/O.

       i. First-come, First-Served
       ii. Round Robin (*aka* preemptive FCFS)
       iii. Shortest Job First
       iv. Shortest Remaining Time First (*aka* preemptive SJF)
       v. Priority
       vi. Priority-based Round Robin (*aka* preemptive Priority)
       vii. Multilevel Feedback Queue Scheduling

   Answer:

   *SJF, SRTF, Priority, Priority-based RR, MLF (the bottom five)*

   (c) (10 points) Say we have the process arrival time chart given below.

Process	Arrival time	Burst Time
p_1	0	16
p_2	0	10
p_3	6	4
p_4	7	6
p_5	8	10

   Draw a Gantt chart to illustrate how these processes would be scheduled using Round Robin (RR) and First-come First-Served (FCS) scheduling, and calculate the waiting time for each process. If when choosing a process to schedule next you could legally choose any of a number of processes, choose the one with the lowest-numbered name; i.e., choose p_i over p_j if $i < j$. *Do not include context-switching time in your chart.* Assume that the quantum is set to 5 time units.
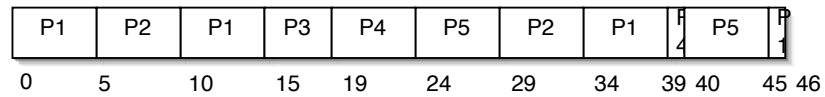
       i. Gantt chart and waiting time for FCFS

   Answer:

P1	P2	P3	P4	P5

   0   16   26   30   36   46
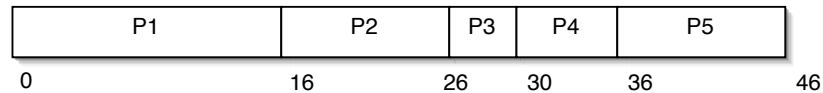
   The wait times for each process are calculated by the formula time of completion - arrival time - burst time. *Therefore: p1: 0, p2: 16, p3: 20, p4: 23, p5: 28*

ii. Gantt chart and waiting time for RR

Answer:

P1	P2	P1	P3	P4	P5	P2	P1	P4	P5	P1

0   5   10   15   19   24   29   34   39 40   45 46

Wait times: p1: 30, p2: 24, p3: 9, p4: 27, p5: 27

(d) (5 points) Round Robin scheduling is said to be better for interactive processes. Explain why this is so.

Answer:

Round robin has better response time, which makes the system more interactive. That is, while SJF may have better average waiting times across all processes, RR ensures that each process never waits more than a certain interval (the quantum times the number of other processes in the system) before being scheduled again.

3

2. (Signals, 15 points)

Below is a program that forks n processes, performs some computation in parallel with them, and finally prints a message as each of them completes.

```
#include <conio.h>

int main(int argc, char **argv) {
  int i, n = atoi(argv[1]);
  int pids[10];
  if (n > 10) n = 10;
  if (n < 0) n = 0;

  for (i=0;i<n;i++) {
    pids[i] = Spawn_Program("foo.exe", "/c/foo.exe", 0);
  }
  /* ... Perform some computation here ... */
  for (i=0;i<n;i++) {
    Wait(pids[i]);
    Print("%d terminated\n",pids[i]);
  }
}
```

This program will reap the spawned child processes in order, even if they don't complete in that order, and only after it finishes performing its computation. This means that the resources of child processes which have terminated may be tied up longer than they need to be.

In project 2 you implemented signals for GeekOS. The signals interface for GeekOS user programs is shown in Figure 1 (on the next page). Rewrite the above program so that it uses signals to reap each child process as it dies, printing the pid of the process that is reaped, as above. (You can just label those parts of the above program that remain unchanged, and refer to that label in your program below so you don't need to recopy anything.)

Answer:

```
    #include <conio.h>
    int n;
    void child_handler(void) {
      int pid, exitCode;
      while ((pid = WaitNoPid(&exitCode)) != -1) {
        Print("%d terminated\n",pid);
        n--;
      }
    }

    int main(int argc, char **argv) {
      int i;
      n = atoi(argv[1]);
      if (n > 10) n = 10;
      if (n < 0) n = 0;
      Signal(child_handler,SIGCHLD);

      for (i=0;i<n;i++) {
        Spawn_Program("foo.exe", "/c/foo.exe", 0);
      }
      /* ... Perform some computation here ... */
      while (n != 0); /* wait til all child processes are reaped */
    }
```

```
/* Signal numbers */
#define SIGKILL  1 /* can't be handled by users */
#define SIGUSR1  2
#define SIGUSR2  3
#define SIGCHLD  4

/* Definition of a signal handler */
typedef void (*signal_handler)(void);

/* Flags for default handlers */
#define SIG_DFL  (signal_handler)0
#define SIG_IGN  (signal_handler)1

/* Sends the signal sig to process pid */
int Kill(int pid, int sig);

/* Registers the handler h for signal sig for the current process.  If
   this process is sent this signal, then h will be called. */
int Signal(signal_handler h, int sig);

/* Wait for child process pid to terminate, returning its exit code. */
int Wait(int pid);

/* Reaps any terminated child process.  Returns -1 if there is no such
   process; otherwise returns the pid of the terminated process and
   stores its exit code in *status. */
int WaitNoPID(int *status);
```

Figure 1: GeekOS signals interface for user processes

3. (Concurrency, 20 points)

(a) (5 points) Properly implementing a concurrent programming system requires balancing the tensions of *safety* and *liveness*. Explain how ensuring safety can compromise liveness. Likewise, explain how ensuring liveness can compromise safety.

Answer:

Safety is a property of objects, and is typically ensured by restricting concurrent access. Liveness is a property of activities (threads), which is ensured by reducing waiting/blocking. Ensuring safety can result at worst in deadlock, which compromises liveness. Ensuring liveness can result in concurrent access to an object, possibly leading to corruption and compromising safety.

(b) (15 points) Suppose that your operating system does *not* support *preemption*; rather, a process/thread can only be context-switched (1) if it blocks to wait on I/O, or (2) if it voluntarily gives up the CPU using the routine `Yield()`.

Below there are three pairs of code snippets, each of which represents the operation of two threads with one shared variable, `g`, which is accessed within a critical section. Recall that a solution to the *critical section problem* must satisfy three criteria: *mutual exclusion*, *progress*, and *bounded waiting*. In each case, indicate whether together the two threads properly implement a critical section delineated by the comments in the code. If not, explain which of the criteria is not met, and why. (You only need to mention one criterion if you think that more than one applies.)

```
Thread 1:
while (1) {
  Yield();
  /* critical section begins */
  g = g + 1;
  g = g - 2;
  /* critical section ends */
}
```

```
Thread 2:
while (1) {
  /* critical section begins */
  g = g + 1;
  g = g - 2;
  /* critical section ends */
}
```

Answer:

This violates progress (and arguably bounded waiting) because once thread 2 starts to run it will never relinquish the CPU to thread 1, which thus will never be able to run its critical section from then on.

Thread 1:
```
while (1)
  Yield();
  /* critical section begins */
  g = g + 1;
  g = g - 2;
  /* critical section ends */
}
```

Thread 2:
```
while (1)
  /* critical section begins */
  g = g + 1;
  g = g - 2;
  /* critical section ends */
  Yield();
}
```

Answer:

*This is correct. (There is an argument that this might violate progress because when thread 1 exits its critical section but has not yet executed **Yield** then thread 2 should be able to enter its critical section. However, thread 2 has not technically asked to enter its critical section yet because it has yielded.)*

Thread 1:
```
while (1)
  Yield();
  /* critical section begins */
  g = g + 1;
  Print("In a critical section!\n");
  g = g - 2;
  /* critical section ends */
}
```

Thread 2:
```
while (1)
  Yield();
  /* critical section begins */
  g = g + 1;
  g = g - 2;
  /* critical section ends */
}
```

Answer:

*This violates mutual exclusion because the **Print** statement in thread 1 may cause it to get context-switched so that thread 2 can simultaneously enter its critical section.*

4. (Deadlock, 25 points)

   (a) (15 points) There are four necessary conditions for deadlock: *mutual exclusion*, *no preemption*, *hold and wait*, *circular waiting*. In the following scenarios, indicate whether or not a deadlock could occur. If so, argue why these four conditions could become true. If not, indicate which of the conditions will never become true.

   i. (5 points) We have $n \geq 2$ processes which access shared resources R_1, R_2, and R_3, and each of these resources has an associated semaphore S_1, S_2, and S_3. To access a resource R_i, each process first calls $P(S_i)$; the process calls $V(S_i)$ when finished. A process may hold any number of resources at a time, and may acquire them in any order.

   Answer:

   A deadlock could occur, because all four necessary conditions are satisfied. The use of semaphores ensures mutual exclusion and no preemption, and the fact that any number of resources can be held and yet are acquired one at a time is tantamount to "hold and wait." Finally, since semaphore operations do not force a fixed order, we have "circular waiting."

   ii. (5 points) We have the same scenario as above, except that a process may only hold one resource at a time.
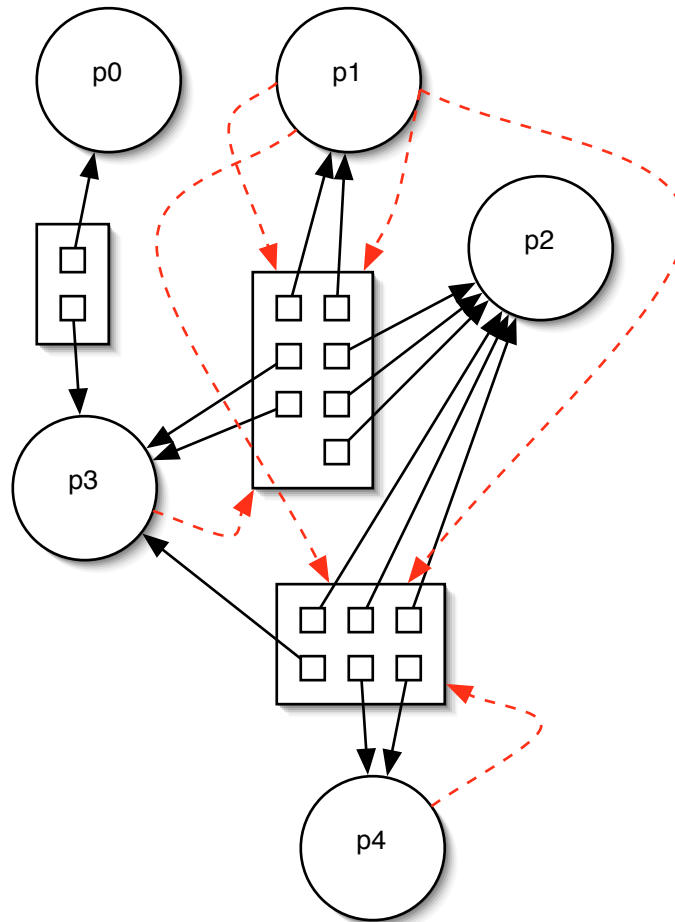
   Answer:

   A deadlock cannot occur because "hold and wait" will never be satisfied. (Note that circular waiting will never be satisfied either, but that is a consequence of no "hold and wait.")

   iii. (5 points) We have the same scenario as (a) above, except that the resources must be acquired in descending order by a given process (i.e., R_3 before R_2, etc.).

   Answer:

   A deadlock cannot occur because "circular waiting" will never be satisfied.

(b) (10 points) The diagram shown below is a resource allocation graph. Does it contain a deadlock? If so, say which processes are involved in the deadlock. If not, indicate a *safe sequence* by which each process could run to termination, assuming that it will not request any more resources. Hint: convert this graph into the *available*, *request*, and *allocation* matrixes, and perform the deadlock detection algorithm.

Answer:

This is not in a deadlocked state. There are many possible safe sequences; e.g.

p_0, p_2, p_1, p_3, p_4

p_2, p_0, p_3, p_1, p_4

p_2, p_1, p_4, p_3, p_0

p_2, p_4, p_1, p_3, p_0

5. (Miscellaneous, 15 points)

(a) (5 points) An operating system is *event-driven*. How does this make it different from a program like a compiler?

Answer:

> *The operating system waits for events to occur (in the form of interrupts) and then acts on them. It runs forever. Conversely a compiler takes a specific input to produce a specific output and then terminates.*

(b) (5 points) General-purpose CPU's like the x86 and PowerPC support *dual-mode operation* for protecting system resources. In particular, dual mode operation can be used to enforce that only the kernel performs I/O operations on behalf of processes. This ensures that resource usage respects *unique use*: that is, that an I/O device is only in use by one process at a time. If the hardware did not support dual-mode operation, could the OS still enforce unique use? Explain why or why not.

Answer:

> *The answer is yes. There are (at least) two ways one could do it:*
> - *Before running a program, the operating system could make sure it didn't contain any instructions for performing I/O (getting this right will be tricky, but doable), to ensure it only does I/O via system calls.*
> - *The operating system could avoid scheduling any new process before the current one terminates. That is, the OS could implement a batch scheduler.*

(c) (5 points) Which of the following components of program state are shared across threads in a multi-threaded process:

  i. Register values
  ii. Heap memory
  iii. Global variables
  iv. Stack memory

Answer:

> *Heap memory and global variables are shared, while each thread has its own stack and copy of the register values.*