50004 - Operating Systems - Lecture 8

Oliver Killane

09/11/21

Lecture Recording

Lecture recording is available here

Lecture Recording

Lecture recording is available here

Deadlock

When two processes mutually block access to a resource the other requires to progress. An example is below:

```
Sema scanner, dvd_writer;

int main(int argc, char **argv)
{
    sema_init(&scanner, 1);
    sema_init(&dvd_writer, 1);
}
```

```
void process_A(void)
{
    sema_down(&scanner);
    sema_down(&dvd_writer);
    scand_and_record();
    sema_up(&dvd_writer);
    sema_up(&scanner);
}
```

```
void process_B(void)
{
    sema_down(&dvd_writer);
    sema_down(&scanner);
    scand_and_record();
    sema_up(&scanner);
    sema_up(&dvd_writer);
}
```

Dining Philosophers

```
#define PHILOSOPHERS 5

Sema chopsticks[PHILOSOPHERS];

int main(int argc, char **argv)
{
    /* Initialise chopstick semaphores. */
    for (int i = 0; i < PHILOSOPHERS; i++) {
        sema_init(&chopsticks[i], 1);
    }

    /* Start philosophers. */
    ...
}
```

```
void eat_and_think(int i)
{
    for (;;) {
        sema_down(&chopstick[i]);
        sema_down(&chopstick[(i + 1) % PHILOSOPHERS]);
```

```
6           /* Eat using chopsticks. */
7           sema_up(&chopstick[i]);
8           sema_up(&chopstick[(i + 1) % PHILOSOPHERS]);
9           /* Think about philosophy for a while. */
10      }
11  }
```

Notice that if every philosopher takes chopstick i at the same time (chopstick to the left), then we have a deadlock. As no one can get the chopstick to the right, so cannot progress.

A set of processes is **deadlocked** if each process is waiting for an event that only another process can cause, and that process is waiting on an event that is directly or indirectly caused by the other.

Resource deadlock is the most common type, and for it 4 conditions must hold:

- **Mutual Exlcusion** Each resoruce is either available or assigned to exactly one process
- **Hold & Wait** Process can request resources while it holds other resources.
- **No preemption** Resources given to a process cannot be forcibly revoked or borrowed.
- **Circular Wait** Two or more processes in a circular chain, each waiting for a resoruce held by the next process.

Resource Allocation Graphs

A **Directed Graph** containing:

A cycle in the graph represents a deadlock.

Deadlock Strategies

- **Ignore**
  The "ostrich algorithm", if contention for resources is low & deadlocks very infrequent, then can iugnore, simply restart when a deadlock occurs.
- **Detection & Recovery**
  After the system becomes deadlocked, detect a deadlock has occured & recover from it (e.g reverting to a saved state).
- **Dynamic Avoidance**
  Dynamically consider every request and decide if it is safe to grant.

  Needs information to determine safety of a request (e.g resoruce use, current owners etc).
- **Prevention**
  Prevent deadlocks by ensuring at least one of the four conditios can never hold.

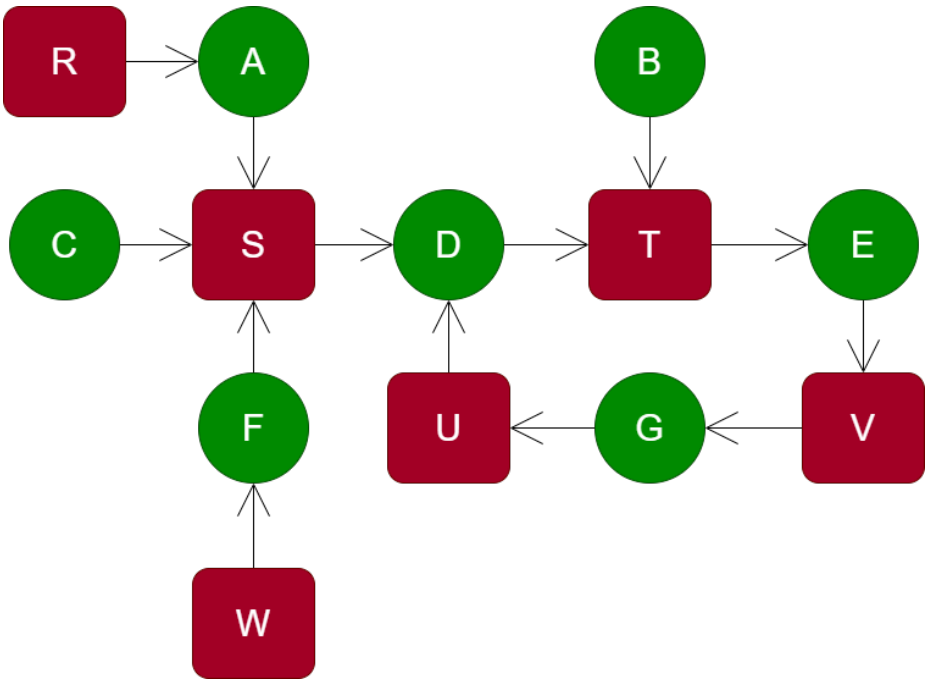
Detection & Recovery

Dynamically builds resource ownership graph and searches for cycles (**Depth-First Search**, when edge inspected it is marked and not visited again).

When cycle detected, recover.

```
# Code to detect a cycle in the resource allocation graph
def traverse(node, visited):
    if node in visited:
        CYCLE!!
    else:
        visited.append(node)
        for outgoing_edge in node:
            if outgoing_edge.unmarked:
                outgoing_edge.mark
                traverse(outgoing_edge.to, visited)

def detect(graph):
    for node in graph:
        traverse(node)
    NO CYCLE!!
```



Here we have a cycle as $D, E\&G$ are in a cycle and cannot gain the resources they need to progress. Further B may be blocked as E holds T.

Recovery

- **Pre-emption**
  Temporarily take a resource from an owner & give it to another.

- **Rollback**
  Processes are periodically checkpointed (memory image, state), rollback to the last state (note we must still prevent deadlock from occuring when this state progresses).
- **Killing Processes**
  Select a random process in the cycle and kill it! (Would work for jobs such as compiling, but not for a DB system - depends if it is easy to restart the process)

Deadlock Prevention

Attack one of the 4 deadlock conditions.

- **Mutual Exclusion**
  Share the resource

  Issue: This risks race conditions.
- **Hold & Wait** Require all processes to request resources before start, if not available then wait.

  Issue: Need to know what resoruces are needed in advance.
- **No-preemption**
  Issue: Potential issues with race conditions, data corruption etc (e.g forcing a printer to give up half way through a print).
- **Circular Wait**
  Force single resource per process - Issue: Not optimal, requires far more resources.

  Index resources, processes wait for resources in order - Issue: For a large number or resources & processes it is difficult to organise.

Communication Deadlock

Ordering resources & careful scheduling will not help here, instead we must alter the communication protocol to use timeouts.

Livelock

Processes/Threads are not blocked, however the program/system as a whole fails to make progress.

For example a busy wait on a acquiring a mutex, when the mutex cannot be acquired.

```
Mutex A, B;

int main(int argc, char **argv)
{
    // start processes:
    ...
}
```

```
void process_B(void)
{
    enter_region(B);
    enter_region(A);
    // Do stuff
    leave_region(A);
    leave_region(B);
}
```

```
void process_B(void)
{
    enter_region(A);
    enter_region(B);
    // Do stuff
    leave_region(B);
    leave_region(A);
}
```

Or a system receiving messages, may never run a lower priority thread under high load (**receive livelock**).